Rapport de Projet MIDL 1

Élimination de quantificateurs dans la théorie DO

Dalla-Costa Valentin & Elissalde Baptiste

22407035 & 22409466

Table des matières

1 Introduction

Ce projet s'inscrit dans le cadre de l'UE PROJET MIDL 1 et porte sur l'implantation d'une procédure de décision automatisée pour la théorie des ordres denses (DO), elle se base sur les connaissances et savoirs que nous avons acquis lors de l'UE Philosophie des sciences en L1.

Le but de ce projet est d'implémenter cette procédure dans un programme capable de déterminer la validité d'une formule logique en utilisant la méthode de l'élimination des quantificateurs. Le problème consiste à transformer une formule close (sans variables libres) contenant des quantificateurs ($\forall$, $\exists$) en une constante logique ($\top$ ou $\bot$).

Pour ce faire, nous traitons successivement chaque variable quantifiée en analysant les relations d'ordre ($<$) et d'égalité ($=$) présentes dans le corps de la formule à l'aide de nombreuses fonctions et surtout d'une représentation de données bien choisie et adéquate que nous explicitons dans la partie `Représentation des données`*.*

Cette procédure s'appuie sur un ensemble d'étapes décrites en annexe du sujet et que nous suivrons tout le long de ce projet. Au cours de ce travail, nous avons dû relever plusieurs défis techniques :

- *La représentation des données : Choisir des structures informatiques appropriées en Python (fondées sur le fichier syntax.py fourni) pour manipuler efficacement les arbres syntaxiques des formules.*
- *Les hypothèses : Mettre en place les fondations de notre programme en créant les premières fonctions permettant à une formule d'être correctement écrite pour être traitée.*
- *Le prétraitement : Mettre en œuvre la transformation des formules en formes normales négatives et disjonctives (NNF et DNF) afin de préparer l'élimination proprement dite.*
- *L'extension arithmétique : Explorer la possibilité d'étendre cet algorithme aux nombres rationnels, permettant ainsi de traiter des expressions linéaires plus complexes (comme $3 * x - z < y$).*

Ce rapport détaille donc notre démarche d'implémentation, les choix de structures de données que nous avons privilégiés pour faciliter le traitement récursif, ainsi que les analyses de chacune des fonctions utilisées dans notre programme.

Vous trouverez donc dans ce rapport détaillé l'ensemble des informations nécessaires à la bonne compréhension de notre programme et des choix que nous avons faits. Allant du manuel d'utilisateur à l'analyse des fonctions.

2 Manuel d'utilisateur

Pour utiliser le programme, assurez-vous d'ouvrir le fichier **DECISION** dans le terminal de commande. Celui-ci doit être composé des fichiers : `syntax.py, fonctions.py, fonctions_test.py et main.py`

1. Lancez une console Python dans ce répertoire.

2. Lancez le programme principal avec la commande ***python3 main.py*** (sous python 3).
3. À partir de là, vous pouvez désormais choisir parmi les trois choix possibles : le 1 : les tests, il vous sera demandé ensuite quels tests exécuter.
4. Vous pouvez aussi directement choisir de lancer le programme principal de la procédure de décision avec en choix n°2 une formule déjà donnée dont le déroulé de la décision vous sera détaillé.
5. Dans le cas du troisième choix, vous entrerez votre formule en suivant les instructions de construction de la formule, celles-ci sont détaillées en annexe. Ensuite, cela reviendra au deuxième choix avec la résolution si possible de la formule à l'aide de la procédure de décision.

3 Description de l'implantation

3.1 Représentation des données

Durant toute la première partie de notre programme, c'est-à-dire des tests des fonctions de base jusqu'à la fin des vérifications des hypothèses, nous utiliserons la représentation classique sous forme de formule de logique.

Notre représentation des données changera donc après cette dernière étape, nous allons donc utiliser à partir de l'élimination des quantificateurs depuis l'intérieur jusqu'à la fin de notre procédure de décision une tout autre représentation.

Nous avons eu une première idée basée sur deux dictionnaires que nous avons approfondie mais nous nous sommes rendu compte que celle-ci était trop compliquée à mettre en place dans la suite du programme, on vous la laisse donc en annexe de ce rapport avant de vous expliciter notre choix final.

Tout d'abord, comme suggéré dans le sujet, nous aurons une liste avec les variables de notre formule qui seront alors chacune associée à un quantificateur existentiel car les hypothèses pour la procédure de décision seront déjà satisfaites à cette étape. Par exemple pour la formule suivante $\exists x \, \exists y \, \varphi$ nous aurons la liste [x,y] dans cet ordre pour pouvoir éliminer les quantificateurs à partir du dernier élément de la liste jusqu'au premier. De ce fait nous pourrons passer en paramètre de certaines de nos fonctions seulement le corps de la fonction, ce qui simplifiera le code de celle-ci.

Cet ordre est simplement pour une meilleure compréhension car en Python nous pouvons accéder facilement au dernier élément d'une liste, sinon nous aurions inversé l'ordre pour l'utiliser comme une pile en ajoutant les variables à partir du début de la formule. De même, la liste ne sera pas exactement celle-ci mais plutôt [sub=QuantifF(q=Ex(), var='x', body=None), QuantifF(q=Ex(), var='y', body=None)], de ce fait nous pourrons rajouter les négations dans cette liste.

Ensuite lorsque nous arriverons à la dernière étape de notre procédé de prétraitement nous devons tirer la quantification existentielle à l'intérieur de nos disjonctions, car à partir de cette étape nous avons une formule qui est sous forme normale disjonctive. De ce fait nous aurons l'apparition d'une deuxième liste qui sera composée pour chacun de ses éléments, des quantificateurs et de leur potentielle négation ainsi que d'une conjonction

de relations. Notre liste sera alors en quelque sorte une disjonction de sous-formules (ici des conjonctions) extraites de notre formule.

$$Liste \leftrightarrow \bigvee_j (\exists x\, \psi_j)$$

Nous pouvons apparenter la virgule entre les termes de la liste comme des disjonctions.

Nous voici maintenant dans ce qui fera office de dernière représentation de notre formule amenant à sa décision complète. Ici nous avons alors pour chaque formule une liste de sous-formules toutes étant un ensemble de conjonctions. Nous allons se concentrer sur une seule de ces sous-formules pour la résoudre.

Nous allons alors pour chaque sous-formule regrouper ses termes. Pour cela nous créons une liste qui sera composée de plusieurs autres listes :

- La première liste sera la liste des quantificateurs de notre sous-formule.
- La deuxième quant à elle sera composée de tout les termes de la forme x $<$ .. avec x une variable choisie et identique pour toutes les listes.
- La troisième dans le même style sera composée des termes de la forme .. $<$ x.
- La quatrième sera les égalités du type x $=$ .. .
- La cinquième elle sera tout les termes de notre sous-formule ne contenant pas la variable x.

Une fois chacune des sous-formules sous cette forme nous pouvons continuer notre traitement pour supprimer la variable x souhaitée en étudiant les différentes listes créées.

3.2 Implantation de la procédure de décision

3.2.1 Hypothèse

Pour l'implantation de la procédure de décision nous avons suivi les instructions de la partie A.2 du sujet. Donc tout d'abord nous avons dû émettre des hypothèses sur notre formule que nous avons traduites par des fonctions créées dans notre code.

Celles-ci ne peuvent être exécutées seulement si la formule est close, c'est-à-dire sans variables libres et qu'elle est composée seulement de symboles relationnels ($<$ et $=$). Nous avons traduit cela dans le code par les fonctions `freeVar(f)`, `isClose(f)`, `toClose(f)`, `isJustSymboleRelationnel(f)` et `isElimPossible(f)` qui les utilisent. Toutes celles-ci sont décrites dans la partie **Analyse des fonctions**.

Une fois toutes les exigences satisfaites nous pouvons commencer la démarche nous permettant de satisfaire toutes les hypothèses nécessaires à la procédure de décision :

- Tout d'abord nous devons convertir la formule en forme prénexe, c'est-à-dire que tous ses quantificateurs sont situés à gauche de la formule, cela empêche d'avoir plusieurs variables de même nom mais de signification différentes. Pour cela nous avons créé la fonction `isPrenexe(f)` qui vérifie seulement si notre formule est sous forme prénexe, cela est supposé mais on l'appellera quand même pour suivre la démarche lors du programme principal.

- Ensuite nous devons convertir les quantificateurs universels en existentiels en suivant la formule suivante : $\forall x.\varphi \leftrightarrow \neg(\exists x.\neg\varphi)$. La fonction `allToExist(f)` suit cette formule et renvoie donc notre formule modifiée.

— Puis pour finir nous pouvons éliminer les quantificateurs à partir de l'intérieur. À partir de cette étape et pour toute la suite de notre programme nous utiliserons le deuxième type de représentation des données, n'étant plus des formules mais des listes. Celle-ci est expliquée plus haut dans la partie **Représentation des données**.

Donc nous allons créer pour nous aider les fonctions `extraireQuantificateurs(f)` et `reconstruireAvecQuantificateurs(f,quantif)` qui respectivement extrairont les quantificateurs de la formule et reconstruiront la formule à l'aide d'une telle liste. Leurs fonctionnements sont expliqués plus bas.

3.2.2 Prétraitement

Dans cette partie nous avons désormais une formule qui remplit toutes les hypothèses et donc nous pouvons procéder à son prétraitement. Pour cela commençons par tirer les négations à l'intérieur de notre formule, le but de cette étape est de nous retrouver avec une formule sous forme normale négative. Pour cela nous avions déjà créé la fonction `nnf(f)` lors de la première partie du projet, nous allons donc la reprendre pour construire notre fonction `tirerNegation(f)`. Nous expliquons son fonctionnement dans la partie **Analyse des fonctions**.

Ensuite, une fois les négations tirées à l'intérieur de la formule, nous pouvons les supprimer le plus possible. Pour cela nous nous aidons des lois données dans le sujet qui sont :

$$(a)\ \neg(z \prec z') \leftrightarrow (z = z' \vee z' \prec z)$$
$$(b)\ \neg(z = z') \leftrightarrow (z \prec z' \vee z' \prec z)$$

à l'aide de celles-ci nous pouvons créer la fonction `elimNegation(f)` qui nous renverra le corps de la formule f avec le moins de négations possible.

De ce fait, nous pouvons transformer notre formule en forme normale disjonctive pour faciliter la procédure de décision. Pour cela nous avons déjà la fonction `dnf(f)` qui prend une formule sans quantificateur et renvoie celle-ci sous forme normale disjonctive. Nous créons donc simplement une fonction `toDisjonctive(f)` qui enlève les quantificateurs de la formule avant d'y appliquer notre fonction `dnf(f)`.

Pour la dernière étape du prétraitement et comme expliqué dans la partie méthodologie nous allons créer une fonction qui nous renvoie une nouvelle liste, celle qui représentera notre forme normale disjonctive sous la forme de sous-formules composées exclusivement de conjonctions. Le but de cette fonction que nous appellerons `tirerQuantif(f)`, est de donner cette forme à notre formule : $\bigvee_j (\exists x\, \psi_j)$. Les termes de cette disjonction étant chacun des éléments de notre nouvelle liste, les ψ_j sont eux un ensemble d'aucune ou de plusieurs conjonctions de relations $u \prec v$ ou $u = v$..

À cette étape nous avons donc fini la partie de prétraitement de notre formule, nous avons obtenu à l'arrivée une disjonction de sous-formules conjonctives que nous allons désormais pouvoir étudier. Cela dans la partie **Suppression de variables**.

3.2.3 Suppression de variables

Dans cette partie nous allons enfin nous focaliser sur la résolution de notre formule, nous rappelons que nous avons obtenu par le biais des parties précédentes et de leurs

modifications sur notre formule générale une liste de sous-formules toutes composées uniquement de conjonctions. Nous pouvons donc désormais traiter chacune d'entre elles à travers de multiples étapes que l'on va vous expliciter.

Tout d'abord commençons par les deux premières étapes qui seront les plus simples, la première est de retirer tous les quantificateurs inutiles de notre formule, c'est-à-dire ceux qui ne concernent aucune variable de notre sous-formule. Par exemple $(\exists y.)$ dans $\exists x.\exists y.x = x$. Pour cela nous créerons la fonction `elimQuantifInutile(f)` qui nous renverra notre liste avec chaque sous-formules démunie de ses quantificateurs inutiles.

Notre deuxième étape sera de rechercher exactement cette relation ($x < x$) dans chacune des sous-formules, si elle est présente alors celle-ci sera modifiée pour devenir : $\bot$.

Par la suite nous devrons utiliser une implémentation et méthodologie différente, celle-ci est comme les autres décrites dans la section Représentation des données.

Nous sommes donc ici dans le cas où il n'y a pas la relation $x < x$ dans notre formule, nous allons donc regrouper en 5 parties les conjonctions de notre formule, comme expliqué dans la section Représentation des données nous aurons la fonction `regrouperTermes(f,x)` qui va créer une liste de 5 listes composées des relations de la forme $x < u$, $u < x$, $x = u$ et les termes où x n'apparaît pas, cela en se basant sur la variable x passée en paramètre. À partir d'ici nous pouvons distinguer trois cas et suivre le schéma proposé dans le sujet.

S'il y a une relation "=" alors nous remplaçons x par sa nouvelle valeur, sinon si nous rencontrons les termes $x < u$ et $v < x$ alors on supprime x en les transformant en $v < u$. Et pour finir si nous avons que des relations $x <$ ou $< x$ alors nous gardons que les termes ne contenant pas x car ces relations n'ont aucun impact sur la formule générale.

3.2.4 Décision d'une formule

Nous voilà donc maintenant dans la dernière partie qui clôture notre programme de décision, celui-ci sera court et consistera simplement à effectuer l'ensemble de nos traitements sur la formule donnée.

Il contient la fonction principale de notre programme `decision(f)` qui sera explicitée dans la partie **Analyse des fonctions**. Le peu de fonctions que contient cette partie consiste à effectuer une suppression de variable à l'aide des fonctions mises en place dans la partie précédente ainsi qu'un enchaînement de celle-ci.

Commençons le fonctionnement de celle-ci, nous allons donc pour chaque conjonction de notre formule de départ effectuer les différents cas mis en place, c'est-à-dire si $x < x$ est présent, ou regrouper les termes pour ensuite faire une distinction de cas. Suite à cela nous nous retrouvons forcément avec une formule réduite car nous avons supprimé une variable et donc un quantificateur de notre formule, mais pas sa négation s'il en avait une attention. Si cette suppression de variable nous suffit à avoir une constante $\top$ ou $\bot$ alors nous aurons fini le programme. Mais comme souvent cela ne suffit pas il est libre à l'utilisateur de faire une deuxième suppression de variable voire plus. C'est là qu'intervient `enchainementSupDeVar(conjonctions)` qui sera la fonction nous permettant cela. Celle-ci est comme un interface qui demande à l'utilisateur si celui-ci veut supprimer une autre variable, c'est aussi ici que nous avons ajouté la fonctionnalité permettant à l'utilisateur de choisir s'il veut le détail de cette suppression car celui-ci est long et n'est pas forcément nécessaire à chaque fois.

Pour finir la fonction `decision(f)` consiste en un ensemble d'appels à ces fonctions que nous avons créées tout le long du projet et qui finit par l'appel à la dernière fonction du projet, `isFormuleValide(list)`. Qui avec la liste de conjonctions représentant notre formule de départ désormais devenues des constantes, à la suite des suppressions de variable, définit si notre formule est valide ou non.

3.3 Analyse des fonctions

Voici la liste des fonctions actuellement implantées dans le fichier (`fonctions.py`) ainsi que leur fonctionnement :

3.3.1 Fonctions de base

- **dual(f)** : Parcourt récursivement l'arbre syntaxique pour inverser les opérateurs $\wedge$ et $\vee$.
- **nnf(f)** : Transforme une formule en Forme Normale Négative en tirant les négations. Pour ce faire nous effectuons des tests sur chaque instance possible, si ce n'est pas une négation alors on applique la fonction sur le corps de la formule et si c'est une négation quatre possibilités apparaissent :

  Si c'est une constante on renvoie son inverse, si c'est une comparaison alors l'égalité devient deux inégalités et ainsi de suite pour si c'est des inégalités. Si c'est une autre négation alors la double négation s'annule puis si c'est une opération booléenne alors le OU devient ET en appliquant des négations sur les deux termes de l'opération.
- **dnf(f)** : Convertit la formule en Forme Normale Disjonctive en utilisant la distributivité. Fonctionne de la même manière que `nnf()`, cette fois ci cela fonctionne récursivement en distribuant soit la conjonction soit la disjonction de sorte que l'on ait une disjonction de conjonctions.
- **freeVar(f)** : Parcourt récursivement l'arbre syntaxique en ajoutant les variables liées dans un ensemble quelle renvoie ensuite.
- **extraireQuantificateurs(f)** : Dans cette fonction itérative nous parcourons la formule dans une boucle "while" comportant 3 tests (il y a un quantificateur, un quantificateur précédé d'une négation ou deux négations à la suite). S'il y a deux négations alors on les supprime et on continue l'algorithme, sinon on ajoute dans une liste créé auparavant le quantificateur précédé ou non d'une négation.
- **reconstruireAvecQuantificateurs(body, quantif)** : Cette fonction plus simple recompose seulement notre formule à l'aide du corps et des quantificateurs, une boucle "for" parcoure la liste de quantificateur à l'envers et ajoute l'élément au début du corps déjà donné.
- **reconstruireAvecTermes(list)** : Dans le même style que `reconstruireAvecQuantificateurs` cette fonction permet de reconstruire une formule sous forme normale conjonctive. La liste passée en paramètre étant composée de listes composées des termes de la formule nous avons simplement à parcourir ces listes et à placer une conjonction entre chaque termes, nous renvoyons ensuite le résultat de `reconstruireAvecQuantificateurs` appliqué au quantificateur étant la première liste de la liste passé en paramètre et à la formule que nous venons de reformer.
- **allVarInFormula(f)** : Cette fonction récursive nous permet de récupérer les variables de la formule. Pour cela nous effectuons comme dans la plupart des autres

fonctions des tests sur le type de notre formule, si celui-ci est autre qu'une comparaison alors on applique la fonction récursivement sur le corps de notre formule, sinon on ajoute dans une liste, si elles y sont pas déjà les variables de notre comparaison.

— **affichage()** : Ces deux fonctions nous servirons dans le programme principal à partir de la dernière partie de notre procédure de décision, c'est à dire lorsque nous manipulons une liste de sous-formules ou une liste de liste de termes. Elles permettent toutes deux un affichage simplifié d'une formule à partir d'une simple liste passé en paramètre. De plus nous avons rajouté la possibilité de rajouter un préfixe passé en paramètre rendant l'affichage plus modulable.

— **extract()** : Ces trois fonctions similaires extracts marchent de la même manière, elles permettent de renvoyer une liste composée seulement des termes d'une seule sorte, égalité, inégalité, ... Pour cela nous "descendons" dans l'arbre syntaxique de la formule jusqu'à tomber sur une relation, si celle-ci est celle recherchée alors on l'ajoute à la liste sinon nous ne faisons rien.

— **input_formula_interactive()** : Pour la création de cette fonction je me suis aidé de chatGPT pour pouvoir avoir la représentation la plus simple et compréhensible possible car il est difficile de rentrer une formule dans un terminal python. De ce fait nous avons créé un système basé sur un ensemble de choix qui se succède.

Tout d'abord un helper `choose(prompt, choices)` affiche un menu (clé → description) et boucle jusqu'à obtenir un choix valide.

Ensuite une fonction récursive build() demande le type de formule (5 choix) puis, selon le choix, demande les informations nécessaires et construit la sous-formule : ("1" Constante : demande True/False, "2" Comparaison : demande identifiant gauche, opérateur ($=$ ou $<$) puis identifiant droit → retourne ComparF(left, Eq()|Lt(), right), ...).

Cela reste quand même une longue tâche de rentrer une formule mais cela est déjà bien plus compréhensible que d'autres solutions.

La description de l'utilisation de cette fonction se trouve en annexe.

3.3.2 Fonctions d'aide pour la procédure de décision

Fonctions de vérification des hypothèses :

— **isClose(f)** : Récupère la liste des variables libres avec **freeVar(f)**, s'il n'y en a pas alors la formule est close sinon elle ne l'est pas.

— **toClose(f)** : Transforme une formule en formule close en ajoutant un quantificateur $\forall$ pour chaque variable libre.

— **isJustSymboleRelationnel(f)** : Parcourt récursivement l'arbre syntaxique de la formule et vérifie si elle est composée seulement de symboles relationnels et non de constante ou de fonction.

— **isElimPossible(f)** : Vérifie si la formule est close et qu'avec des symboles relationnels à l'aide des fonctions précédentes.

— **isPrenexe(f)** : On vérifie que la formule soit sous la forme prénexe à l'aide d'une fonction qui renvoie si oui ou non une formule contient des quantificateurs. Tant que la formule commence par un quantificateur on appelle récursivement sur son contenu et sinon le reste de la formule ne doit pas contenir de quantificateur.

— **allToExist(f)** : Cette fonction se base seulement sur cette formule ($\forall x.\varphi \leftrightarrow \neg(\exists x.\neg\varphi)$) et l'applique à la formule donnée.

Fonctions de prétraitement :

— **tirerNegation(f)** : Comme nous l'avons dit précédemment nous allons nous aider de la fonction **nnf(f)** pour cette fonction là. Pour cela le seul enjeu est de donner la formule sans quantificateurs à notre sous fonction **nnf(f)**, ce qui se fait simplement avec la fonction **extraireQuantificateurs(f)**. Une fois fait nous pouvons appliquer **nnf(f)**.

— **elimNegation(f)** : En utilisant les formules de négation cette fonction parcourt l'arbre syntaxique de f pour remplacer le corps d'une négation en suivant la formule si cela est possible. Nous avons donc 2 cas pour les 2 implications de celles-ci. Plus un cas pour la double négation que l'on simplifiera.

— **toDisjonctive(f)** : Tout comme pour **tirerNegation(f)** nous avons simplement à séparer le corps de la formule de ses quantificateurs avec **extraireQuantificateurs(f)** puis nous appliquons notre fonction **dnf(f)**. À cette fonction sera associé **isDisjonctive(f)** qui vérifie si notre formule est bien sous forme normale disjonctive.

— **isDisjonctive(f)** : Parcoure notre formule et vérifie si celle-ci est bien une disjonction de conjonctions. Autrement dit qu'il n'y a pas de disjonction dans une suite de conjonctions et que les deux côtés d'une disjonction sont des formes normales disjonctives.

— **tirerQuantif(f)** : Cette fonction nous est très importante car elle nous permet de séparer notre formule générale en plusieurs sous-formules qui seront plus simples à décider.

De plus nous partons d'une forme normale disjonctive donc il nous suffit dans cette fonction de récupérer chaque sous-formule qui n'est pas composée de disjonction. Pour cela on fait appel à une forme itérative de fonction basée sur un algorithme similaire au parcours d'un arbre binaire en largeur avec une liste.

De ce fait, on commence par vérifier si notre formule est une disjonction, si c'est le cas on ajoute ces deux côtés dans notre liste de disjonctions, sinon on ajoute directement la formule entière. Puis nous lançons une boucle qui parcoure notre liste de disjonctions en traitant chaque élément de celle-ci, si c'est encore une disjonction alors on ajoute de nouveau dans la même liste, sinon on ajoute la conjonction ou simplement la relation dans la liste que nous renverrons.

Fonctions de suppression de variable :

— **elimQuantifInutile(conjonctions)** : Dans cette fonction nous allons créer une autre fonction qui nous renverra la liste des variables présentes dans notre formule (`allVarInFormula(f)`), ce qui nous permettra ensuite de supprimer les quantificateurs inutiles.

Ici toute la difficulté est dans les négations, si un quantificateur est inutile alors on le supprime, rien de compliqué. Mais si celui-ci est précédé d'une négation alors on l'applique sur le quantificateur suivant.

Dès lors nous avons 3 cas : Si celui-ci n'a pas déjà de négation alors on la décale devant celui-ci, sinon on supprime la négation (car deux à la suite) et dans le dernier cas, si c'était le dernier quantificateur alors on décale cette négation au corps de la formule. Pour tous ces tests on s'est aidé du booléen `isNot` qui correspond au manque ou non d'une négation. En aucun cas lors de cette fonction nous supprimons une négation car cela modifierait le sens logique de la formule.

— **searchXltX(list)** : Cette fonction récursive parcourt notre formule et recherche si la relation x < x est présente, pour cela nous faisons un appel récursif sur le corps d'une négation d'un quantificateur ou des sous-formules d'une opération, jusqu'à atteindre une relation où nous vérifions si celle-ci est de la forme x < x.

— **regrouperTermes(f,x)** : Cette fonction très importante se décompose assez facilement en un ensemble de fonctions très semblables sur des relations. En effet pour cela nous avons créés trois fonctions **extractultx, ..xltu et ..eqx** qui nous permettent très simplement de récupérer les relations correspondantes, celles-ci sont décrites dans la partie `Fonction de bases`.

Une fois ces relations récupérées nous les plaçons à la suite dans une liste précédée des quantificateurs. Pour finir nous avons créé la fonction récursive interne **removeExtractedTerms(f)** qui marche de la même manière que extract mais qui renvoie la liste seulement composée des relations sans la variable x, soit les relations qui ne sont pas déjà dans la liste. Nous avons donc à la fin de cette fonction notre formule sous forme de liste avec les relations regroupées selon leur genre.

— **xeqw(f,x)** : Après avoir regroupé les termes nous pouvons supprimer la variable x, cette fonction traite de la cas de la présence de la relation x = u. Une fois la variable u récupérée nous parcourons tout les termes de notre liste afin de remplacer les occurrences de x par u et dans le même temps supprimer les relations u = u qui apparaîtront.

— **simplifierInegalites(list,list)** : Cette fonction traite le cas où il y a les relations x < u et v < x dans notre formule. Elle prend en paramètre la liste des relations x < et < x afin de les simplifier. Pour cela nous allons simplement parcourir les deux listes avec deux boucles imbriquées, pour chacune des relations du type (x < u) et (v1 < x, v2 < x, ...) nous allons ajouter à notre nouvelle liste les relations (v1 < u, v2 < u, ...). Nous aurons alors supprimé toutes les occurrences de x dans notre formule.

3.3.3 Fonctions de décision d'une formule

— **supDeVariables(f,bool)** : Comme supposé dans le nom cette fonction nous permet de supprimer une variable dans une conjonction. Pour se faire nous allons vérifier si notre formule n'est pas déjà une constante ou si celle-ci contient la relation x < x à l'aide de la fonction `searchXltX`, ces cas étant les plus simples à traiter.

Si ce n'est pas le cas nous regroupons les termes comme expliqué plus haut dans la partie `Implantation de la procédure de décision`. Ce suit alors un enchaînement de tests nous permettant de supprimer la variable "x". Cette fonction comporte une grosse partie d'affichage pour décrire les étapes de la suppression de variable.

De ce fait nous avons rajouté la possibilité de supprimer cet affichage grâce au booléen passé en paramètre en bloquant la sortie standard.

— **enchainementSupDeVar(list)** : Cette fonction est composée essentiellement d'inputs qui demandent à l'utilisateur si celui-ci veut supprimer une variable ou bien afficher le détail de la suppression de variable. Cela prend beaucoup de place et

c'est pour cela que nous avons créés une fonction à part entière pour cela, car pour chaque input nous devons faire une boucle "while" pour s'assurer que l'utilisateur rentre une réponse correcte. Tout cela dans une boucle "while" générale pour si l'utilisateur continue à exécuter des suppressions de variable

— **isFormuleValide(list)** : Cette fonction assez simple est composée simplement d'une boucle qui effectue sur chaque sous-formule un test, si celle-ci est $\top$ alors on renvoie True car si une sous-formule d'une disjonction est vraie alors la formule entière l'est, sinon on passe à la sous-formule suivante, si il n'y a aucun $\top$ alors on renvoie False.

— **decision(formule)** : Voici notre fonction générale de décision de formule celle-ci est assez simple et permet simplement de soulager le main. Elle est composée de plusieurs étapes correspondant à celles suivies tout le long du projet.

Tout d'abord avec les fonctions, `toClose, isClose, isElimPossible, allToExist` et `isJustSymboleRelationnel` nous vérifions les hypothèses et nous préparons la formule à être traité.

Ensuite nous effectuons le prétraitement de notre formule, nous exécutons à la suite sur notre formule les fonctions `tirerNegation et elimNegation` permettant de supprimer les négations sur le corps de la formule puis `toDisjonctive et tirerQuantif` pour séparer notre formule en sous-formules sous formes normales conjonctives.

Pour finir nous retirons les quantificateurs inutiles et appelons la fonction `enchainementSupDeVar` afin de supprimer les variables pour simplifier la formule. À la fin de tout cela nous appelons `isFormuleValide` pour vérifier la validité de notre formule.

3.3.4 Fonction de tests et main

— **Fonctions de tests** : Dans le dossier `DECISION` nous pouvons trouver un fichier `fonctions_tests.py`, celui-ci comporte l'ensemble des fonctions de tests du programme. En effet chacune des fonctions utilisées dans le programme est testée par un ou plusieurs tests de sorte à couvrir les cas qui pourraient potentiellement poser problème.

Elles sont toutes écrites approximativement de la même manière avec des formules déjà écrites passées en paramètre de la fonction puis un affichage permettant de vérifier si le résultat est bien celui attendu.

De plus nous retrouvons à la fin de ce fichier un suite de cinq fonctions qui regroupent selon leur type les fonctions de tests, hypothèse, suppression de variable, ... De ce fait nous avons à la fin de notre fichier la fonction `test_global()` qui effectue en fonction de la demande de l'utilisateur les fonctions de tests associées.

— **Main** : Toujours dans ce même dossier nous retrouvons le fichier `main.py` qui contient le programme à exécuter de notre projet.

Celui-ci est en grande partie composé d'inputs de l'utilisateur vis-à-vis du programme que celui-ci veut lancer. Il a alors trois choix, les tests, le programme sur une formule déjà donnée et le programme sur une fonction rentrée par l'utilisateur. À la fin de cela le programme main.py exécute alors le programme associé, c'est-à-dire `test_global(), decision(formule)` ou bien `input_formula_interactive()` puis `decision(formule)`.

4 Annexe

4.1 Première idée de représentation des données

Avant l'arrivée de la représentation de nos sous-formules avec une liste composée de 5 listes de termes. Nous avons eu l'idée d'une représentation assez similaire basée sur des dictionnaires.

Voici donc un extrait de notre ancienne explication vis-à-vis de cette idée de représentation :

Elle sera composée d'une liste et de deux dictionnaires dont on va vous expliquer l'utilité :

— Tout d'abord comme suggéré dans le sujet nous aurons une liste avec les variables de notre formule qui seront alors chacune associée à un quantificateur existentiel car les hypothèses pour la procédure de décision seront déjà satisfaites à cette étape. Par exemple pour la formule suivante $\exists x\, \exists y\, \exists z\, \varphi$ nous aurons la liste [x,y,z] dans cet ordre pour pouvoir éliminer les quantificateurs à partir du dernier élément de la liste jusqu'au premier. Cet ordre est simplement pour une meilleure compréhension car en Python nous pouvons accéder facilement au dernier élément d'une liste, sinon nous aurions inversé l'ordre pour l'utiliser comme une pile en ajoutant les variables à partir du début de la formule. De même, la liste ne sera pas exactement celle-ci mais plutôt [sub=QuantifF(q=Ex(), var='x', body=None), QuantifF(q=Ex(), var='y', body=None), QuantifF(q=Ex(), var='z', body=None)], de ce fait nous pourrons rajouter les négations dans cette liste.

Ensuite nous avons pensé à une suite de cette représentation qui sera possible juste après avoir tiré les négations à l'intérieur de notre formule. Nous aurons besoin de deux dictionnaires pour cela :

— Le premier dictionnaire que nous appellerons dicoEqual sera composé de toutes les égalités de la formule ou sous-formule étudiée.

— Dans le même style que le premier nous aurons le dictionnaire dicoLeft qui sera composé de toutes les inégalités de la formule ou d'une sous-formule.

Pour les cas simples ces deux dictionnaires suffisent mais dès lors que nous aurons des formules constituées de $\vee$ et de $\wedge$ nous devrons séparer notre formule en sous-formules pour les résoudre les unes après les autres. Lorsque nous rencontrons un $\wedge$ nous pouvons mettre l'égalité ou l'inégalité dans le même dictionnaire mais si nous avons un $\vee$ il faut alors extraire les deux sous-formules et créer les deux dictionnaires pour chacune afin de résoudre cela.

4.2 Utilisation de input_formula_interactive()

Voici dans cette annexe une courte explication de l'utilisation de la fonction `input_formula_interactive()` définie dans le fichier `fonctions.py`.

Cette fonction permet à l'utilisateur de construire une formule étape par étape en faisant des choix dans le terminal.

Voici donc la liste des choix possibles ainsi que le déroulement une fois ceux-là choisis :

1. — Constante : demande True/False → retourne ConstF.
2. — Comparaison : demande identifiant gauche, opérateur ($=$ ou $<$) puis identifiant droit → retourne ComparF(left, Eq()|Lt(), right).
3. — Négation : construit récursivement la sous-formule et retourne NotF(sub).
4. — BoolOp : construit récursivement gauche et droite, choisit $\wedge$ ou $\vee$ → BoolOpF(left, Conj()|Disj(), right).
5. — Quantificateur : choisit $\forall$ ou $\exists$, lit la variable liée puis construit récursivement le corps → QuantifF(All()|Ex(), var, body).

Tous les choix de menu sont vérifiés : les noms des variables comme les chiffres entrés lors du choix. Pour finir la fonction affiche la formule construite et la retourne.

Remarque : la construction est récursive, donc vous pouvez imbriquer autant de sous-formules que nécessaire.